Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2011-016 March 24, 2011

SEEC: A Framework for Self-aware Management of Multicore Resources

Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

SEEC: A Framework for Self-aware Management of Multicore Resources

Henry Hoffmann[1] Martina Maggio[1,2] Marco D. Santambrogio[1,2] Alberto Leva[2]
Anant Agarwal[1]
[1]*Computer Science and Artificial Intelligence Laboratory MIT*
{*hank, mmaggio, santa, agarwal*}*@csail.mit.edu*
[2]*Dipartimento di Elettronica e Informazione, Politecnico di Milano*
{*maggio, santambr, leva*}*@elet.polimi.it*

Abstract

This paper presents SEEC, a self-aware programming model, designed to reduce programming effort in modern multicore systems. In the SEEC model, application programmers specify application goals and progress, while systems programmers separately specify actions system software and hardware can take to affect an application (e.g. resource allocation). The SEEC runtime monitors applications and dynamically selects actions to meet application goals optimally (e.g. meeting performance while minimizing power consumption). The SEEC runtime optimizes system behavior for the application rather than requiring the application programmer to optimize for the system. This paper presents a detailed discussion of the SEEC model and runtime as well as several case studies demonstrating their benefits. SEEC is shown to optimize performance per Watt for a video encoder, find optimal resource allocation for an application with complex resource usage, and maintain the goals of multiple applications in the face of environmental fluctuations.

1 Introduction

Modern computing systems have greatly increased the burden on application programmers. In addition to expertise in an application domain, developers must have the systems knowledge required to design applications that meet multiple competing goals (e.g. high performance and low power) and maintain these goals in fluctuating environments with varying workloads and unreliable resources.

To help reduce this burden, we propose a new programming model targeting modern multicore processors. In the SElf-awarE Computing (SEEC) model, illustrated in Figure 1, application developers use an application programming interface (API) to indicate the application's goals and current progress, while systems developers use a dedicated system programming interface (SPI) to describe the actions (or adaptations) that the system software and hardware implement. The SEEC runtime meets application goals optimally using an adaptive control system, which learns application and system models online allowing rapid response to environmental fluctuations and ensuring optimal adaptation while avoiding local minima.

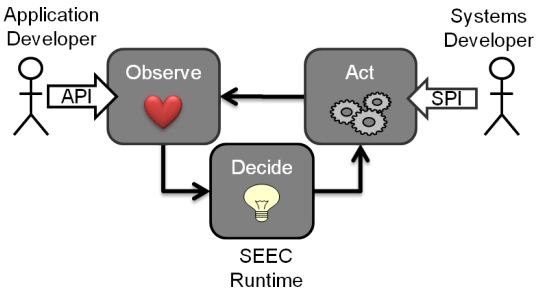


Figure 1: The SEEC model.

1.1 Example

Consider the development of a video encoder whose goal is to encode thirty frames per second while minimizing power. Furthermore, these goals must be met even though different videos (and even frames within one video) differ in their compute demands and these demands cannot be predicted a priori. In a traditional system, the application developer must understand the power and performance tradeoffs of different system configurations (such as number of cores, clock speed, and memory usage) and optimize the encoder to meet performance with minimal power while adapting to both input and system fluctuations.

In contrast, SEEC's runtime system observes application behavior and optimizes the system for the application. Using SEEC, the encoder developer indicates the goal of thirty frames per second and the current speed of the encoder. Independently, systems developers specify actions that affect applications (e.g. allocation of cores, clock-speed, and memory). SEEC's runtime decision engine determines a sequence of actions that achieve thirty frames per second while minimizing power. If an input becomes more difficult, the encoder does not meet its goals and SEEC assigns it additional resources. If an input becomes less difficult, the encoder exceeds its goals, and SEEC will reclaim resources to save power. In addition, SEEC continuously updates its internal models of applications and systems, so it can adapt if new resources become available or if existing resources fail.

1.2 Background

The SEEC programming model builds on previous work in self-aware computing [23, 26]. Such systems have var-

iously been called adaptive, autonomic, self-tuning, self-optimizing, self-*, etc; and have been implemented in both software [37] and hardware [2]. Self-aware systems are characterized by the presence of *observe-decide-act* (ODA) loops.

Researchers have explored general approaches where phases of the ODA loop can be customized [11, 19, 28, 35, 46]. These approaches generally focus on the application layer and all customization is done by a developer working at that layer. For example, ControlWare [46] supports customization of observation and action at the application level; the application programmer is responsible for providing both feedback and adaptations. This focus on the application level limits the available actions to those the application developer knows and can directly control. Similarly, self-aware implementations focusing on the system level (e.g. hardware adaptation [2, 7, 12]) require that observations and actions be specified in terms of system parameters and prevents system level adaptations from using application specified feedback or simultaneously coordinating the actions of both applications and system.

In contrast, the SEEC model decouples the specification of observation, action, and decision steps by establishing separate interfaces for these phases. This separation allows different individuals working at different layers of the system to concentrate on the most appropriate phase for their expertise and reflects the classical process control paradigm where experts who build actuators (e.g. pumps and valves) need not know about the system in which they will be used (e.g. a water treatment plant). Using SEEC, application programmers expect the system to adapt to meet goals but need not understand what actions to take or how to implement them, reducing programmer burden. Similarly, system-level actions are specified without understanding the feedback mechanism that drives adaptation or having to infer application performance from low-level metrics. Furthermore, by receiving direct feedback from applications, the SEEC runtime system guarantees its decisions have a positive impact on application goals. In addition, by decoupling the specification of observation and adaptation, SEEC is able to coordinate actions to maintain goals for multiple applications simultaneously, something application-level self-aware approaches do not support.

1.3 Evaluation

The SEEC model and framework is implemented as a runtime system and set of libraries for Linux, which we evaluate in several case studies. To demonstrate the generality of the approach, we show how the SEEC runtime can control the PARSEC [6] benchmarks through resource allocation and we show that such dynamic resource management produces fewer performance errors than static resource allocation. In the next study, the SEEC runtime optimizes the performance per Watt of a video encoder across a range of inputs. By dynamically adapting resource allocation to fluctuations in video input, SEEC is able to outperform an oracle that statically allocates resources at program launch. Next, SEEC's ability to learn system models online is tested by controlling an application with a complex relationship between performance and allocated resources. By using system and application feedback simultaneously, SEEC is able to avoid local minima and meet performance requirements while reducing total system power by 15%. Finally, SEEC is used to maintain performance of multiple applications while responding to an unexpected loss of compute power. By simultaneously adapting system and application resources, SEEC is able to maintain application goals despite a 33% drop in processor frequency.

1.4 Contributions

This paper makes the following contributions:

- It presents a self-aware programming model where phases of observe-decide-act loops are specified independently, each by the developer whose expertise is most appropriate for the task at hand. This model separates application and system concerns, allowing the system to optimize itself to meet application needs.
- It describes a self-aware decision engine based on a generic, adaptive second-order control system, capable of adapting applications, system software, and hardware resources while updating its internal models online in response to environmental changes.
- It describes how this general, decoupled approach to self-aware computing can be used to coordinate adaptations and maintain the goals of multiple applications simultaneously.
- It shows the generality of the approach by modifying the PARSEC benchmarks to work in the SEEC model.
- It evaluates the SEEC model in a number of case studies, demonstrating SEEC's ability to optimize competing goals and respond to varying workloads and other unforeseen events.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the SEEC model and runtime system. Section 4 presents several case studies evaluating the SEEC framework and the paper concludes in Section 5.

2 Related Work

A self-aware, adaptive, or autonomic computing system is able to alter its behavior in some beneficial way without the need for human intervention [20, 23, 26, 37].

Table 1: Comparison of several self-aware approaches.

	ControlWare [46]	Tunability Interface [11]	Agilos [28]	Choi & Yeung [12]	Bitirgen et al. [7]	SEEC
Observation	Application	System	System	System	System	Application & System
Decision	Control*	Classifier	Control	Hill Climbing	Neural Network	Adpative Control
Action	Application	Application	Application	System	System	Application & System

Self-aware systems have been implemented in both hardware [2, 7, 12] and software [37]. Some example systems include those that that manage resource allocation in multicore chips [7], schedule asymmetric processing resources [40, 36], optimize for power [25], and manage cache allocation online to avoid resource conflicts[45]. In addition, languages and compilers have been developed to support adapting application implementation for performance [43, 3], power [4, 39], or both [18]. Adaptive techniques have been built to provide performance [5, 28, 34, 38, 46] and reliability [9] in web servers. Real-time schedulers have been augmented with adaptive computing [8, 14, 30]. Operating systems are also a natural fit for self-aware computation [10, 22, 24, 33].

Researchers have developed several general self-aware approaches that can be customized without requiring expertise in adaptive system design. Such systems include: ControlWare [46], Agilos [28], SWiFT [13], the *tunability interface* [11], AutoPilot [35], and Active Harmony [19]. These approaches focus on customization at the application level. For example, ControlWare allows application developers to specify application level feedback (such as the latency of a request in a web server) as well as application level adaptations (such as admission control for requests). Unfortunately, these approaches do not allow application level feedback to be linked to system level actions performed by the hardware, compiler, or operating system. In contrast, SEEC allows applications to specify the feedback to be used for observation, but does not require application developers to make decisions or specify alternative actions (application developers can optionally specify application-level actions, see Section 3.2). Thus, SEEC allows application programmers to take advantage of underlying system-level adaptations without even knowing they are available.

Other researchers have developed self-aware approaches to adapt system level actions. Such system-level approaches include machine learning hardware for managing a memory controller [21], a neural network approach to managing on-chip resources in multicores [7], a hill-climbing technique for managing resources in a symmetric multithreaded architecture [12], a number of techniques for adapting the behavior of super-scalar processors [2], and several operating systems with adaptive features [10, 22, 24, 33]. While these approaches allow system level adaptations to be performed without input from the application programmer, they suffer from other drawbacks. First, application performance must be inferred from either low-level metrics (like performance counters [2]) or high-level metrics like total system throughput [7], and there is no way for the system to tell if a specific application is meeting its goals. In addition, these systems are more difficult to extend. For example, if a new resource becomes available for allocation a neural network, (e.g. [7]), will have to be redesigned and reimplemented. In contrast, SEEC allows systems developers to specify available actions independently from the metrics that will be used to evaluate their effectiveness. In addition, SEEC can combine actions specified by different developers and learn models for these new combinations of actions online.

Table 1 highlights the differences between SEEC and some representative prior efforts. The table includes several general approaches for specifying application level adaptation and several approaches for specifying system level adaptation for resource management. For each project the table shows the level (system or application) at which observation and actions are specified and the methodology used to make decisions.

In contrast with systems shown in Table 1, SEEC is a general approach for resource management allowing applications to specify how they are to be observed, while system software and hardware (or, optionally, applications themselves) specify available actions. Observations are specified using an interface to indicate application performance and goals. Actions are specified using a dedicated interface to indicate available actions (such as allocation of system resources) along with coarse estimates of their costs and benefits. An adaptive control system determines the actions to take given current observations of both the application and the system. Adaptive control allows the SEEC runtime system to combine actions specified by different developers and learn the interactions of these actions online. In addition, the adaptive control system allows SEEC to respond quickly to fluctuations in the underlying environment.

As noted in the table, several approaches to building self-aware systems use control theory within their decision engines. Hellerstein et al [16] and Karamanolis et al [22] have both suggested that control systems can be used as "off-the-shelf" solutions for managing the complexity of modern computing systems, especially multi-tiered web-applications. While these authors note that existing control solutions can be used, their development requires identification of a feedback mechanism

Table 2: Roles and Responsibilities in the SEEC model.

Phase	Applications Developer	Systems Developer	SEEC Runtime Infrastructure
Observation	Specify application goals and performance	-	Read goals and performance
Decision	-	-	Determine how much to speed up the application
Action	-	Specify actions and a function that performs actions	Initiate actions based on result of decision phase

and translation of an existing control model into software. This leads to solutions that address a specific computing problem using control theory, but do not generalize [29, 34, 41, 42]. In contrast, the SEEC control system does not solve a specific problem, but provides a general control strategy using a widely applicable feedback mechanism and thus overcomes some limitations of prior approaches as described in [15]. In addition, SEEC's generalized control approach is, itself, adaptive, allowing SEEC to automatically update its decision engine online in response to unforeseen events such as changes in available resources, workload fluctuations, or even the emergence of new adaptations.

3 SEEC Framework

There are three distinct roles in the SEEC model: application developer, system software developer, and the SEEC runtime infrastructure. Table 2 shows the responsibilities of each of these three entities for the three phases of closed loop execution: observation, decision, and action. The application developer indicates the application's goals and current progress toward those goals. The systems developer indicates a set of actions and a function which implements these actions[1]. The SEEC runtime system coordinates actions to meet goals.

SEEC's runtime maintains goals using the closed loop system illustrated in Figure 2. As shown in the figure, SEEC implements three separate adaptation levels (AL 0-2), where AL 1 and 2 are optional. At adaptation level 0, SEEC implements a standard, model-based feedback control system. Adaptation level 1 augments this basic system by incorporating an online model of application behavior, allowing SEEC to quickly detect fluctuations in both application workload and the underlying computing infrastructure. Adaptation level 2 further improves SEEC's decision making by performing online estimation of the benefits and costs of all actions and sets of actions. AL 2 allows SEEC to combine actions specified by different developers and detect changes or errors in action costs and benefits online; this is done automatically without intervention by the application or system programmer. Each increase in adaptation level provides greater flexibility and a (small) increase in overhead.

[1]For clarity, we distinguish between the role of application and systems developer, but in practice these can be filled by the same person.

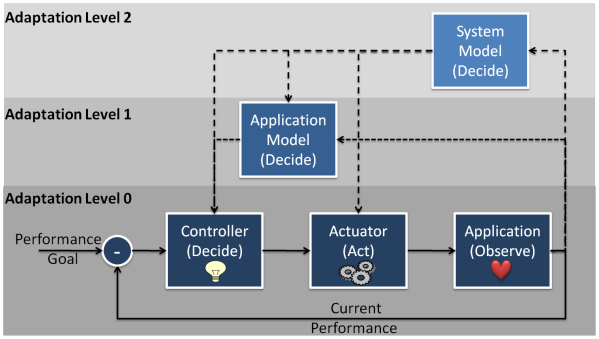


Figure 2: SEEC block diagram.

3.1 Observe

The SEEC model provides an API application programmers use to indicate application goals and progress, i.e., how the application is to be observed. This API is an extension of the Application Heartbeats interface [17], and its key abstraction is a heartbeat. Applications use a function to emit heartbeats at important intervals, while additional API calls allow the specification of performance goals in terms of a target heart rate or a target latency between specially tagged heartbeats.

In addition to raw performance data specified as heartbeats, application programmers can specify several other properties. Programmers can indicate a sliding window over which to calculate average heart rate or heartbeat latency. Using a window serves as a low-pass filter on heartbeat data and smoothes out both noise and systematic variations in performance. This window is specified at the application level because the application developer has more knowledge of expected heartbeat variations than the rest of the system. In addition, application programmers can inform SEEC of preferred tradeoffs.

The SEEC model assumes that increasing application performance generally comes at some cost (e.g., an increase in power consumption). Therefore, SEEC allows application developers to indicate preferred tradeoffs. Currently SEEC supports two tradeoff spaces: application-level tradeoffs and system-level tradeoffs. Indicating a preference for one tradeoff over another directs SEEC to exhaust all actions affecting the preferred tradeoff space before attempting any actions in the second. For example, if system-level tradeoffs are preferred, then the SEEC runtime will only use actions specified at the application level if the target performance cannot be

met by any combination of system-level actions. This interface is extensible so more tradeoffs can be specified as more tradeoff spaces are explored.

3.2 Act

The SEEC model provides a separate, system programmers interface for specifying actions that can be taken in the system. A set of actions is defined by the following features: an identifier for each action, a function which implements the corresponding action, and an array of the estimated costs and benefits in the tradeoff space. The performance benefits of an action are listed as speedups while the tradeoffs are listed as increased costs. By convention, the action with identifier 0 is considered to be the one with a speedup of 1 and a cost of 1; the speedup and costs of additional actions are specified as multipliers. Additionally, the systems developer specifies whether an action can affect all applications or a single application; in the case of a single application, the developer indicates the process identifier of the affected application. Finally, for each action the systems developer indicates a list (possibly empty) of conflicting actions. Conflicting actions represent subsets of actions which cannot be taken at the same time; e.g. allocation of both five and four cores in an eight core system.

For example, to specify the allocation of cores to a process in an eight core system, the developer might specify eight actions with identifiers $i \in \{0, ..., 7\}$. To implement these actions, the developer specifies a function that takes a process identifier and action identifier i, and then binds the process to $i + 1$ cores. The systems developer provides an estimate of the increase in performance and power consumption associated with each i. For the core allocator, the speedup of action i might be $i + 1$, i.e. linear speedup, while the increase in power consumption will be found by profiling the target architecture. For each action i, the list of conflicting actions includes all j such that $j+1+i+1 > 8$. Finally, the core allocator will indicate that it can affect any application. In contrast, a runtime system that adaptively changes an application's algorithm (e.g. [18]) will only affect executables linked against that runtime.

These models of speedup and cost only serve as initial estimates and the SEEC runtime system can adapt to even large errors in the values specified by the systems developer. However, SEEC allows these models to be specified to provide maximum responsiveness in the case where the models are accurate. SEEC's runtime adjustment to errors in the models is handled by the different adaptation levels and is described in greater detail in the next section.

SEEC combines n sets of actions $A^0, ..., A^{n-1}$ defined by (possibly) different developers using the following procedure. First, SEEC creates a new set of actions where each action in the set is defined by the n-tuple $< a_i^0, a_j^1, ..., a_k^{n-1} >$, and corresponds to taking the ith action from set A^0, the jth action from set A^1, etc. The speedup of each action in the new set is computed as $s_{<a_i^0,...,a_k^{n-1}>} = s_{a_i^0} \times ... \times s_{a_k^{n-1}}$ and the cost is computed similarly. Once the speedup and cost have been estimated for each combined action, SEEC computes the subset of Pareto-optimal actions. In practice, focus on the Pareto-optimal actions greatly reduces the search space of possible actions and maintains optimality. In some cases SEEC may need to combine some actions that affect a single application with others that can affect all applications. In this case, SEEC computes and maintains a separate set of actions for each application. Combining actions specified by multiple developers may result in inaccurate models, and adaptation levels 1 and 2 handle these cases.

3.3 Decide

SEEC's runtime system dynamically selects actions specified by system programmers to meet the goals specified by application programmers. Many features of the SEEC runtime are customizable and can be tuned for the particular characteristics of a specific deployment.

As illustrated in Figure 2, SEEC implements a feedback control system with multiple adaptation levels (AL). Each increased level of adaptation adds additional flexibility at the cost of additional overhead as summarized in Table 3. AL 0 implements a basic, model-based feedback control system. AL 1 turns this system into an adaptive controller by adding a dynamic model of application performance. AL 2 dynamically updates the system models of action costs and benefits. If the application behavior is well understood, and the system models are accurate, AL 0 is sufficient for responsive performance. If application performance is variable, but the system model is reliable and without local minima, then AL 1 will achieve the best performance. If the application is variable and the system models are inaccurate, unreliable, or vary greatly from application to application, then AL 2 is the most appropriate.

3.3.1 Adaptation Level 0

In its most basic form, the SEEC runtime system implements a basic, model-based feedback control system [16], which complements and generalizes the control system described in [31]. The controller reads the performance goal g_i for application i, collects the heart rate $h_i(t)$ of application i at time t, computes a speedup $s_i(t)$ to apply to application i at time t, and then translates that speedup into a set of actions based on the model provided by the systems programmer. SEEC uses a generic second order control system which can be customized

Table 3: Characteristics of Adaptation Levels in SEEC.

	Adaptation Level 0	Adaptation Level 1	Adaptation Level 2
Models learned online	None	Application	Application & System
Response to Application Workload Variations	Slow	Fast	Fast
Response to Errors/Local Minima in System Models	Suboptimal	Suboptimal	Optimal
Overhead	Low	Medium	High

for a specific system by fine-tuning the tradeoff between responsiveness and rejection of noise.

SEEC's controller observes the heartbeat data of all applications and assumes the heart rate $h_i(t)$ of application i at time t is

$$h_i(t) = \frac{s_i(t-1)}{w_i(t-1)} + \delta h_i \tag{1}$$

Where $w_i(t)$ is the *workload* of application i. Workload is defined as the expected time between two subsequent heartbeats when the system is in the state that provides the lowest speedup, i.e. when the system takes action 0. At adaptation level 0, SEEC assumes that the workload is not time variant and any noise or variation in the system is modeled with the term δh_i, representing an exogenous disturbance in the measurement of the heartbeat data for application i.

SEEC's goal is to eliminate the *error* $e_i(t)$ between the heart rate goal g_i and the observed heart rate $h_i(t)$ where $e_i(t) = g_i - h_i(t)$. SEEC reduces $e_i(t)$ by controlling the speedup $s_i(t)$ applied to application i at time t. SEEC employs a generic second order transfer function so users can customize the transient behavior of the closed loop system shown in Figure 2. Since SEEC employs a discrete time system, we follow standard practice [27, p17] and analyze its transient behavior in the Z-domain:

$$F_i(z) = \frac{(1-p_1)(1-p_2)}{1-z_1} \frac{z-z_1}{(z-p_1)(z-p_2)} \tag{2}$$

where $F_i(z)$ is the Z-transform of the closed-loop transfer function for application i and $\{z_1, p_1, p_2\}$ is a set of customizable parameters which alter the transient behavior of the system. The gain of this function is 1, so $e_i(t)$ is guaranteed to reach 0 for all applications. From Equation 2, the generic SEEC controller is synthesized following a classical control procedure [27, p281] and SEEC calculates $s_i(t)$ as:

$$\begin{aligned} s_i(t) &= F \cdot [\, A\, s_i(t-1) + B\, s_i(t-2) + \\ &\qquad C\, e_i(t)\, w + D\, e_i(t-1)\, w\,] \\ A &= p_1 z_1 + p_2 z_1 - p_1 p_2 - 1 \\ B &= -p_2 z_1 - p_1 z_1 + z_1 + p_1 p_2 \\ C &= p_2 - p_1 p_2 + p_1 - 1 \\ D &= (p_1 p_2 - p_2 - p_1 + 1) \cdot z_1 \\ F &= (z_1 - 1)^{-1} \end{aligned} \tag{3}$$

To customize the generic controller for specific behavior, the values $\{z_1, p_1, p_2\}$ must be fixed. For stability, SEEC requires $|p_1|, |p_2| < 1$. Setting $z_1 = z_2 = p_1 = 0$ eliminates transient behavior[2] allowing the system to reach $e_i(t) = 0$ as quickly as possible; however, this formulation is sensitive to noise and changes in δh_i will result in commensurate changes in the applied speedup (see Equation 1). If $p_1 \leq z_1 \leq p_2$, the heart rate slowly converges to g_i. As z_1 approaches p_1, the system will converge more slowly, but will reject larger disturbances in the δh_i term; i.e., in noisier systems z_1 should be closer to p_1.

SEEC translates a set of speedups $\{s_i(t) | \forall i\}$ determined by the control system into a set of actions (using the actions specified by the systems programmer). SEECs actuator has two challenges: first, it must convert the continuous speedup signal $s_i(t)$ into a speedup that can be realized in the discrete action domain; second, it must resolve any conflicting actions for different applications (e.g. assignment of more than the available number of cores). We describe each of these issues in turn.

For each i, SEEC converts $s_i(t)$ into a set of actions using time division output; i.e. by computing two actions to take over a quantum of τ time units. SEEC first searches through the set of Pareto-optimal actions (see Section 3.2) to find an action a with the smallest speedup s_a such that $s_a \geq s_i(t)$. By focusing on Pareto-optimal actions, SEEC ensures a is the lowest cost action whose speedup exceeds the target. Next, SEEC searches the set of Pareto-optimal actions to find an action b with the largest speedup such that $s_b < s_a$. Given these two actions, SEEC executes a for τ_a time units and b for τ_b time units where:

$$\begin{aligned} \tau \cdot s_i(t) &= \tau_a \cdot s_a + \tau_b \cdot s_b \\ \tau &= \tau_a + \tau_b \end{aligned} \tag{4}$$

When working with multiple applications, the control system may request speedups whose realization results in resource conflicts (e.g., in an eight core system, the assignment of 5 cores to one application and 4 to another). SEEC's actuator resolves conflicting actions using a priority scheme. Higher priority applications get first choice amongst any set of actions which govern finite resources. Once a plan is created for a higher priority application, those actions are removed from consideration for lower priority applications. In the example, the higher priority application would be assigned 5 cores with the other

[2] In a control-theoretic sense transient behavior cannot be fully eliminated, but this formulation makes the transient period as small as possible.

forced to use three and find speedup from an additional source if available. If applications have the same priority, SEEC accounts for this by acting as if one is higher for one time quantum and then acting as if others are higher for subsequent quanta.

3.3.2 Adaptation Level 1

Adaptation level (AL) 1 extends SEEC's basic functionality with an adaptive control system which estimates application workload online. This capability allows SEEC to rapidly respond to sudden changes in application performance (e.g. an input video becomes more difficult to encode). In practice, the true workload cannot be measured online as it requires running the application with all possible actions set to provide a speedup of 1, which will likely fail to meet the application's goals. Therefore, SEEC views the true workload as a hidden state and estimates it using a one dimensional Kalman filter [44].

The true (and hidden) workload for application i at time t is represented as $w_i(t) \in \mathbb{R}$ and characterized as

$$\begin{aligned} w_i(t) &= w_i(t-1) + \delta w_i \\ h_i(t) &= \frac{s_i(t-1)}{w_i(t-1)} + \delta h_i \end{aligned} \tag{5}$$

where δw_i and δh_i represent noise in the true workload and heart rate measurement, respectively. Given this model of true workload, SEEC recursively estimates the workload for application i at time t as $\hat{w}_i(t)$ using the following Kalman filter formulation:

$$\begin{aligned} \hat{x}_i^-(t) &= \hat{x}_i(t-1) \\ p_i^-(t) &= p_i(t-1) + q_i \\ k_i(t) &= \frac{p_i^-(t)s_i(t-1)}{[s_i(t)]^2 p_i^-(t) + o_i} \\ \hat{x}_i(t) &= \hat{x}_i^-(t) \\ &\quad + k_i(t)[h_i(t) - s_i(t-1)\hat{x}_i^-(t)] \\ p_i(t) &= [1 - k_i(t)s_i(t-1)]p_i^-(t) \\ \hat{w}_i(t) &= \frac{1}{\hat{x}_i(t)} \end{aligned} \tag{6}$$

Where q_i and o_i represent the workload variance and heart rate variance, respectively. $h_i(t)$ is the measured heartrate for application i at time t and $s_i(t)$ is the applied speedup (according to Equation 3). $\hat{x}_i(t)$ and $\hat{x}_i(t)^-$ represent the a posteriori and a priori estimate of the inverse of application i's workload at time t. $p_i(t)$ and $p_i^-(t)$ represent the a posteriori and a priori estimate error variance, respectively. $k_i(t)$ is the *Kalman gain* for the application i at time t[44].

By estimating workload, SEEC can rapidly detect changes in the difficulty of an application's input (e.g. higher complexity frames in a video encoder) and respond by changing resource allocation appropriately. Thus, adaptation level 1 greatly increases the speed with which SEEC can adapt to unforeseen changes in the environment.

$w_i(t)$ and $s_i(t)$ have an inverse relationship in Equation 1. Therefore, Equation 6 allows SEEC to respond to changes in both application behavior and system resources, as any error in the speedup models will be perceived (at adaptation level 1) as an error in workload and compensated accordingly. For example, suppose the actions available to SEEC are allocation of cores. Further, suppose an application i is meeting its goals with four cores until one of the four is powered down (for temperature reasons). The change in compute power will change the heart rate $h_i(t)$ at time t, which will in turn immediately affect the workload estimate $\hat{w}_i(t)$ (Equation 6) and cause a corresponding change in the applied speedup $s_i(t+1)$ (Equation 3). This change in speedup will result in the allocation of additional cores. If the temperature cools and all cores are restored, the workload estimator will return lower values of $w_i(t)$ and SEEC will reduce the allocated cores.

3.3.3 Adaptation Level 2

AL 1 augments SEEC with a powerful adaptation capability, but it suffers from some short-comings. As discussed above, at AL 1, SEEC is unable to distinguish between an error in the application model $\hat{w}_i(t)$ and an error in the speedup models provided by the systems developer. While AL 1 allows fast adaptation, it can also suffer from using sub-optimal states. For example, consider a system which allocates cores and an application which achieves linear speedup for up to 4 cores, but then achieves no additional speedup due to limits in parallelism. AL 1 cannot distinguish this lack of parallelism from a change in the application's workload, so SEEC might allocate more resources than necessary. AL 2 addresses this problem by incorporating a system model that estimates the true costs and benefits of all actions online on a per application basis. In addition to adding flexibility, AL 2 allows SEEC to adjust to any errors in the action models provided by systems developers. The added flexibility of AL2 comes at a cost of increased overhead as the SEEC runtime system performs more computation to learn system models online.

As with the application workload, the true speedup and cost provided by any given action state is hidden, i.e., it cannot be directly measured without serious disruption to the performance of the running application. Therefore, SEEC once again employs a Kalman filter to estimate the true costs and benefits of the available actions[3]. SEEC

[3]For space reasons, this presentation focuses on estimation of benefits; a filter to estimate costs is almost identical except it measures the cost (e.g. power) instead of the benefit (e.g. speedup).

computes the expected heart rate of application i at time t as a function of the action a. This information is represented as a vector $r_i(t) \in \mathbb{R}^{n\times 1}$, where n is the total number of available actions, and the ath component of the vector $r_i(t,a) \in \mathbb{R}$ is the heart rate associated with action a.

SEEC's model of the system behavior is:

$$\begin{array}{rcl} r_i(t) & = & r_i(t-1) + \delta r_i \\ h_i(t) & = & T_i(t-1) \cdot r_i(t) + \delta h_i \end{array} \quad (7)$$

Where $h_i(t)$ is the measured heart rate of application i at time t and $T_i(t) \in \mathbb{R}^{1\times n}$. T is constructed such that the ath component of T is τ_a/τ, the bth component is τ_b/tau, and all other components are 0. τ_a, τ_b, and τ are related to the time quantum and found using Equation 4.

SEEC computes an estimate $\hat{r}_i(t) \in \mathbb{R}^{n\times 1}$ of the true heart rate as a function of action using an n-dimensional Kalman filter:

$$\begin{array}{rcl} \hat{r}_i^-(t) & = & \hat{r}_i(t-1) \\ P_i^-(t) & = & P_i(t-1) + Q_i \\ u_i(t) & = & T_i(t-1)\ P_i^-(t)\ [T_i(t-1)]^T \\ K_i(t) & = & P_i^-(t) \times T_i(t-1)^T \times \\ & & [u_i(t) + R_i]^{-1} \\ \hat{r}_i(t) & = & \hat{r}_i^-(t) + K_i(t) \times \\ & & [h_i(t) - T_i(t-1)\hat{r}_i^-(t)] \\ P_i(t) & = & [I - K_i(t)\,T_i(t)]\,P_i^-(t) \end{array} \quad (8)$$

Where $Q_i \in \mathbb{R}^{n\times n}$ and $R_i \in \mathbb{R}$ represent the application noise covariance and heart rate measurement noise variance, respectively. $h_i(t)$ is the measured heartrate for application i at time t. $\hat{r}_i(t)$ and $\hat{r}_i(t)^-$ are vectors representing the a posteriori and a priori estimate of the application i's heart rate at time t; the ath element of each vector is the estimate of the heart rate associated with action a. $P_i(t), P_i^-(t) \in \mathbb{R}^{n\times n}$ represent the a posteriori and a priori estimate error covariance, respectively. $K_i(t) \in \mathbb{R}^{n\times 1}$ is the Kalman gain at time t[44]. $u_i(t) \in \mathbb{R}$ is a temporary value used to make the formulae easier to read. We note that $T_i(t)$ has only two nonzero elements, so all matrix updates are rank-2. Thus the Kalman filter update for each application is $O(1)$ and can be implemented efficiently in practice.

Given $\hat{r}_i(t)$ as calculated by Equation 8 SEEC estimates the speedup associated with each action as $\hat{s}_i(t,a) = \hat{r}_i(t,a)/\hat{r}_i(t,0)$ and $\hat{s}_i(t) \in \mathbb{R}$. Where $\hat{s}_i(t,a)$ represents the speedup SEEC estimates application i will achieve at time t by taking action a. At adaptation level 2, SEEC substitutes $s_a = \hat{s}_i(t,a)$ and $s_b = \hat{s}_i(t,a)$ when computing a plan for then next time quantum using Equation 4.

SEEC uses an almost identical Kalman filter formulation to estimate the cost of system actions (omitted for brevity). Adding the ability to estimate system cost and benefits online gives SEEC additional flexibility at the cost of some overhead. At adaptation level 2, SEEC can detect situations where actions are not benefitting an application or where the cost outweighs the benefits, allowing SEEC to recompute the Pareto-optimal actions online and customize them for an individual application or even an individual input. AL 2 also allows SEEC to recover from errors in the system model provided by the systems programmer, combine two sets of actions specified by separate systems developers, or adjust if the system model changes online due to failure or other unforeseen circumstances.

4 Evaluation

This section describes several experiments that evaluate the generality, applicability, and effectiveness of the SEEC programming model. We begin by describing the experimental platform we use to evaluate the model. Next, SEEC is used to control the performance of all applications in the PARSEC benchmark suite [6]. Then, SEEC is used to optimize the performance/Watt of the x264 video encoder benchmark for 16 input videos, each with differing compute demands. The next study demonstrates SEEC's ability to estimate system models online by co-optimizing compute and memory resources for a benchmark that needs the right proportion of both resources. The final case study demonstrates how SEEC can control multiple applications and coordinate application and system level adaptations to maintain performance in a fluctuating computing environment. For each case study, we describe the goals of the application, the system actions available, and the behavior of the SEEC runtime system.

4.1 Experimental Platform

All experiments are run on a Dell PowerEdge R410 server with two quad-core Intel Xeon E5530 processors running Linux 2.6.26. The processors support seven power states with clock frequencies from 2.394 GHz to 1.596 GHz. The `cpufrequtils` package enables software control of the clock frequency (and thus the power state). These processors also support simultaneous multithreading (SMT), or hyperthreads, but we disable this feature for this paper. This machine has two memory controllers and we have installed the `numa` library to manipulate the processor's use of memory controllers. Power is measured by a WattsUp device which samples and stores power at 1 second intervals [1]. All benchmark applications run for significantly more than 1 second so the sampling interval should not affect results. The maximum and minimum measured power ranges from 220 watts (at full load) to 80 watts (idle), with a typical idle power consumption of approximately 90 watts. The SEEC programming model is implemented on this platform as a set of C libraries and a runtime system.

We account for SEEC's runtime overhead by measuring the time it takes to make a new decision, which requires calculating a speedup, selecting actions, and possibly updating the application and system models. On the target platform, AL 0 sustains 39.22 million decisions per second (d/s), AL 1 sustains 18.83 million d/s, and AL 2 sustains 8.87 million d/s. In practice there are other overheads, including signaling the heartbeat and taking the specified actions, but these will be implementation specific. Given the decision rates measured here, the SEEC runtime is unlikely to be the bottleneck.

4.2 SEEC and the PARSEC Benchmarks

This case study demonstrates the broad applicability of SEEC by modifying the PARSEC benchmarks to work in the SEEC model and controlling their performance using system specified actions. Each PARSEC benchmark is made to emit heartbeats as described in [17]. We record the performance of each application on our target platform using all eight cores and the highest CPU frequency; each benchmark requests a target heart rate between 45% and 55% of this maximum speed. Each benchmark is launched with eight threads and the default inputs for the PARSEC native input set.

Using SEEC's systems programmer interface, we independently specify two sets of actions available to the system. First, we specify one set of seven actions corresponding to changes in processor clock frequency; these actions are implemented by a callback function which uses `cpufrequtils`. Second, we specify a set of eight actions, each of which assigns the corresponding number of cores to an application. This set of actions is implemented via a function which changes the processor affinity of all threads in an application. The initial speedup model for each application is simply linear speedup for both clock and core changes, i.e., changing either resource by some factor changes speedup by the same factor. The initial power model for each set of actions is derived by measuring the power of an application that simply does a busy loop of floating point arithmetic. For clock frequency changes, we measure power using a single core. For core changes, me measure power at the highest frequency setting.

We measure the SEEC runtime's ability to maintain application goals with the specified system actions. We run each application with SEEC and record its heart rate at every heartbeat to compute the average of the squares of the distances between the measured heart rates and the desired heart rates (i.e., the integral of the squared error, ISE). For comparison, we run each benchmark without SEEC, but with a static allocation of half the maximum clock speed (1.995 GHz) and half the maximum number of cores (4).

The results of this experiment are shown in Figure 3.

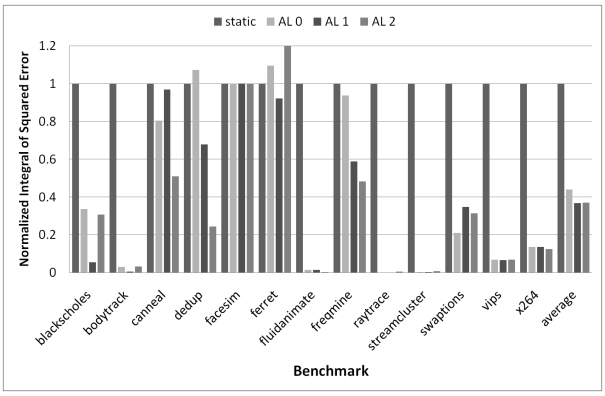


Figure 3: PARSEC with SEEC (lower is better).

The benchmarks (including the average over all benchmarks) are denoted on the x-axis. The y-axis shows ISE normalized to the value measured for the static allocation of half of all CPU resources. Results are shown for the static allocation of resources and for each of the three adaptation levels (AL 0, 1, 2). Figure 3 illustrates that SEEC can control all of the PARSEC applications. On average, AL 0 reduces the performance error by a factor of 2.28, while AL 1 and AL 2 reduce error by a factor of over 2.7. The higher adaptation levels are able to further reduce error by detecting phases in the application and adjusting SEEC's models; e.g. detecting applications which do not exhibit linear speedup with added cores, like x264. For 12 of the 13 benchmarks, at least one of the three adaptation levels provides better control than static resource allocation. The exception is the facesim benchmark for which static allocation and all three adaptation levels execute without control errors.

This study demonstrates several characteristics of the SEEC model. First, PARSEC consists of a variety of important multicore benchmarks, and SEEC's ability to control these benchmarks shows the framework is applicable to a broad range of applications. Adding Heartbeats to the PARSEC benchmarks is straightforward requiring each benchmark be augmented with 5-10 lines of code [17]. Second, this study demonstrates that self-aware monitoring and controlling of an application's resources usage does a better job of maintaining performance goals than static resource allocation. Third, this study demonstrates the benefits of using adaptive control in self-aware systems as both adaptation level 1 and 2 provide better control than adaptation level 0. We note that for all results SEEC is able to maintain performance goals without any prior knowledge of the application. Instead, SEEC uses only heartbeat data observed during the application's execution.

4.3 Optimizing Video Encoding

This case study demonstrates the use of the SEEC model to perform constrained optimization in a fluctuating en-

vironment. Specifically, we use SEEC to help a video encoder meet its performance goals while minimizing power consumption for a variety of input videos, each with differing compute demands. The x264 video encoder benchmark from PARSEC is modified to emit heartbeats as described above, and the encoder requests a heart rate of thirty frames per second.

For this study, additional modifications are made so that when the encoder detects a heart rate of less than 25 frames per second it "drops" the current frame, skipping its encoding and moving to the next frame. For this case study, we again specify two sets of system actions: clock frequency changes and core allocation. The initial models and implementations are as described in the previous section. We then run the encoder on 16 different input videos and evaluate SEEC's ability to maintain the target performance while minimizing power consumption. For each input video, the encoder is initially assigned one core set to the lowest clock speed.

We evaluate SEEC's ability to maintain performance by measuring the number of dropped frames for each input video and recording the fraction of frames which are encoded (not dropped). We then divide this fraction by the average power consumed during the encoding of this input to create a performance per Watt metric. This performance metric rewards the system for reaching the target performance, but provides no extra benefit for exceeding the goal, as appropriate for systems with real-time goals like video encoding. As a point of comparison we construct an oracle which knows the best (and worst) static allocation of resources for each specific input video. The oracle is constructed by explicitly encoding each video with all 56 possible configurations of cores and clock speed. We record the best and worst performance per Watt for each of these static allocations and compare these values to those produced when the encoder is under the control of the SEEC runtime which can dynamically adjust resource allocation.

We note that no single static assignment of resources is best for all inputs. For example, with blue_sky.yuv the best static assignment of resources is 3 cores at maximum clock speed, while for ducks_take_off_1080p.yuv, the best static assignment is 6 cores at maximum clock speed. The fact that there is no single static assignment that is best for all inputs shows the difficulty of this optimization problem. Furthermore, this is a common problem for video encoders as they will routinely be confronted with previously unseen inputs.

Figure 4 shows the results of this case study for each of SEEC's three adaptation levels. The x-axis shows each input (with the average over all inputs shown at the end). The y-axis shows the performance per Watt for each input normalized to the best static assignment. For each input, the first bar represents the worst static assignment,

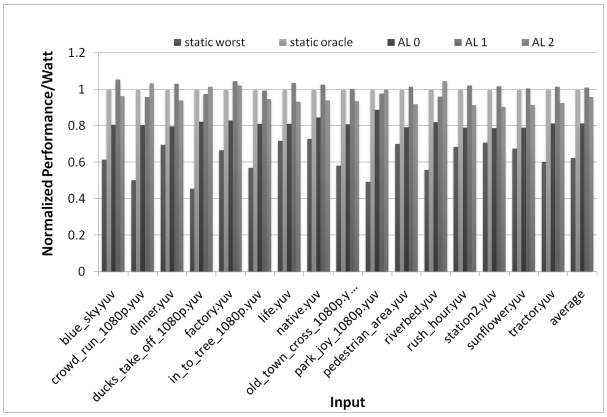


Figure 4: x264 performance per Watt (higher is better).

the middle bar represents the oracle, or best static assignment, and the last three bars represents the performance per Watt for x264 run with each of SEEC's three adaptation levels.

Figure 4 illustrates the benefits of the SEEC approach, and especially the higher adaptation levels. At AL 0, SEEC achieves, on average, about 80% of the performance/Watt of the static oracle. While this is a good figure, AL 0 is limited in the speed with which it can adapt to changes in the video inputs. Changes in the input difficulty can be modeled by changes in the value of the workload in Equations 1 and 3. Adaptation level 0 assumes workload is fixed so the speed of its adaptation is limited.

In contrast, adaptation level 1 actively estimates the true value of the workload online using Equation 6. This adaptation allows AL 1 to outperform the static oracle by a small amount (about 1%) on average. By estimating application workload online, SEEC is able to tailor resource allocation for inputs whose needs vary during execution. For example, the middle section of the PARSEC native input is easier to encode than the beginning and the end. During this section, AL 1 of the SEEC runtime system detects an increase in performance and is able to reduce the amount of resources assigned to the encoder while still meeting its goal.

At AL 2, SEEC achieves about 95% of the static oracle, slightly under-performing AL 1. This difference in performance comes from two factors. First, AL 2 executes more computation, which negatively impacts power consumption without directly increasing encoder performance. Second, in practice the Kalman filter used in AL 2 takes longer to converge than that used in AL 1.

The video encoder on our target platform is CPU bound; adding additional resources always adds speedup. The true value of adaptation level 2 becomes apparent when working with more difficult optimization problems, especially ones where adding resources can slowdown the application. We examine one such application

in the next section.

The results with the video encoder show several properties of the SEEC framework. First, they demonstrate how SEEC reduces programmer burden The video programmer only has to use the Heartbeat interface to declare goals and performance and the SEEC runtime system optimizes the behavior of the program, even tailoring that behavior to specific inputs. With static resource allocation schemes, the video programmer is responsible for profiling and understanding resource management within the system. Additionally, this study shows the optimality of the SEEC framework. Even though no single static assignment is best for all videos, the SEEC runtime is able to adapt its behavior to find an assignment that is close to or better than the best static assignment for each input. Finally, this experiment shows the adaptability of the SEEC framework as videos with multiple regions of differing needs can be allocated the optimal amount of resources for each region.

4.4 Estimating Models Online

This case study demonstrates SEEC's ability to estimate true system models online, even in the case of applications which have complicated responses to resource allocation and where different systems developers specified different sets of actions. Specifically, this study explores SEEC's ability to dynamically allocate resources to meet the needs of the STREAM benchmark [32].

While the STREAM benchmark itself is quite simple, it has an interesting response to resource allocation. STREAM is typically thought of as a memory bound benchmark, but it needs sufficient computational resources before its performance becomes memory limited. For example, on our test platform running STREAM with a single core and two memory controllers is slower than running it using a single core and a single memory controller. As a further example, at 1.6 GHz and two memory controllers, 4 cores achieve higher performance than 5, 6, or 7, but eight is again faster. Overall, there are many such local minima, where adding additional resources slows down the application. This study explores SEEC's ability to avoid these local minima.

For this study, we modify the STREAM benchmark to emit heartbeats. STREAM has an outer loop which repeatedly runs several different tests, and we place the heartbeat signal in this outer loop. We use several different goals for this case study. To establish these goals, we first measured STREAM's performance running on our system with every possible configuration of cores, clock speed and memory controllers, 112 possible states (8 cores, 7 clock speeds, 2 memory controllers). We then established four separate goals for the application: *Min*, equivalent to the performance achieved with one core and one memory controller at the lowest speed; *Median*, the median value of all performance measurements; *Max*, equivalent to the highest observed performance; and *Max+*, equivalent to twice Max and unachievable on the target machine.

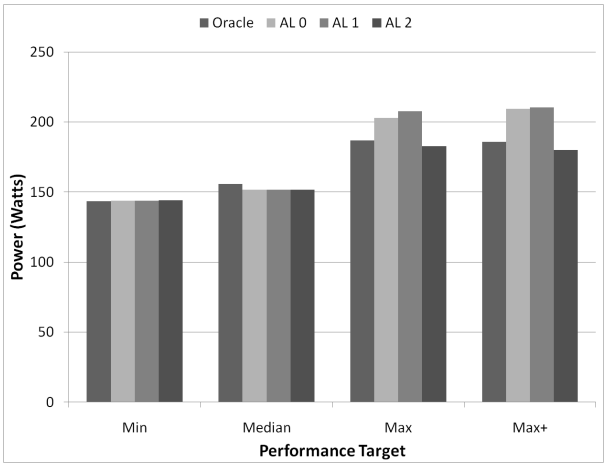
Table 4: ISE for STREAM.

	Min	Median	Max	Max+
AL 0	0.00034	0.012	0.32	39.27
AL 1	0.00034	0.0036	0.096	37.33
AL 2	0.00031	0.0063	0.78	43.70

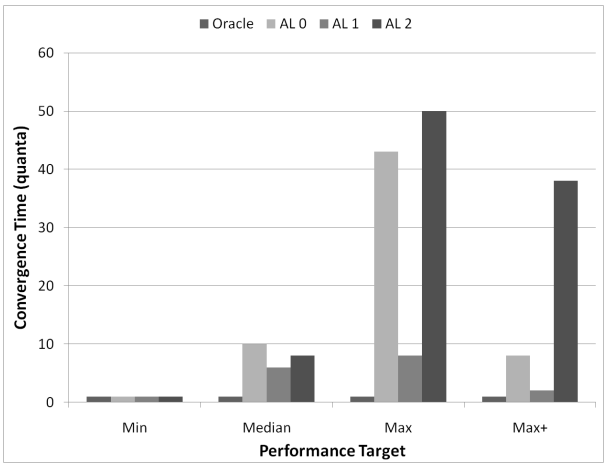
For this case study we make three sets of actions available to SEEC: the core and clock speed actions used in previous sections and a set of two actions which assign memory controllers to the application. We implement these actions by changing the binding of pages to memory controllers using the `numa` interface. The initial model provided to SEEC assumes that the speed increases linearly with the number of memory controllers. We create the model for power by running a separate memory bound application (which repeatedly copies a 1 GB array) and measuring the power consumption using all eight cores at the maximum clock speed and varying the number of memory controllers.

As described in Section 3.2, SEEC's runtime will create an initial model by combining actions from each of the three sets. For this case study, the initial model is optimistic in two respects. First, it assumes linear speedup for all resources. For example, it assumes that doubling the number of memory controllers and the number of cores will quadruple the speed of the application, which is not the case. Second, SEEC's initial model has no local minima, which again is not the case in this instance. This case study evaluates SEEC's ability to overcome these limitations.

For each performance target, we start the STREAM application with the minimum amount of resources, so the SEEC runtime will be responsible for determining the optimal allocation of resources to meet the target. For the first three targets, all adaptation levels converge to the desired value as shown by the ISE scores in Table 4. For the last target, SEEC cannot reach it using the available actions, but all adaptation levels converge to the maximum performance. We evaluate SEEC's ability to adjust its models online by measuring both the power consumption of the system and the speed of convergence for each of the four performance goals. The power consumption data is shown in Figure 5(a) while the convergence time data is shown in Figure 5(b). In both figures, the performance target is shown on the x-axis, while power and convergence time are shown on the y-axes. Power is measured in Watts and represents the full system power, including the overhead of running the SEEC system. Convergence time is the amount of time required for the application to achieve at least 95% of its target performance and is measured in units of SEEC's



(a) Power (lower is better).



(b) Convergence Time (lower is better).

Figure 5: SEEC controlling STREAM.

scheduling quanta.

Figures 5(a) and 5(b) show the tradeoffs available at different adaptation levels. The fastest convergence time is consistently achieved by AL 1; however, this adaptation level over-provisions resources (using all cores at the highest clock speed and both memory controllers), thus it consumes more power than necessary to meet the Max and Max+ performance targets. In contrast, AL 2 has a slower convergence time, but saves power. By updating its system models online using Equation 8, AL 2 is able to determine the true speedups associated with combinations of actions. Thus, AL 2 avoids both local minima and over-provisioning of resources. For example, the maximum performance is achieved using all eight cores and both memory controllers. With that resource allocation, additional clock speed increases power consumption without increasing performance. Only AL 2 is able to detect this condition and avoid increasing the clock speed; AL 2 saves over 30 Watts of total system power for the Max+ target compared to AL 1. Of course this power savings comes at an increase in convergence time compared to AL 1.

This case study demonstrates SEEC's ability to estimate the true values of the system models online. At AL 2, SEEC is able to combine models for different sets of actions while still achieving optimal resource allocation. In practice, there is a tradeoff between the responsiveness of the system and the optimality of SEEC's decision engine. By using different, optional adaptation levels, SEEC allows this tradeoff to be customized as desired.

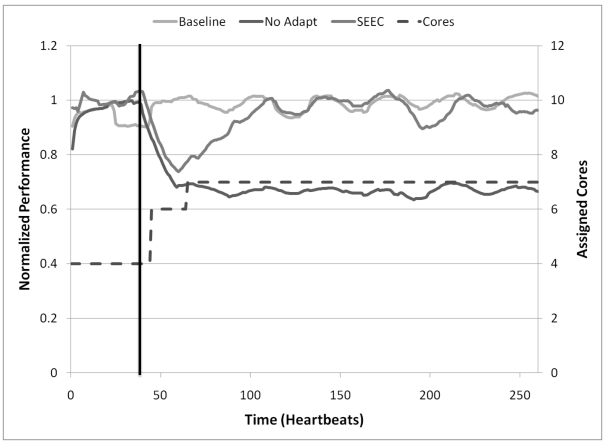
4.5 Managing Multiple Applications

In the final case study, the SEEC runtime system maintains the goals of multiple applications as the underlying hardware fluctuates. In addition, the SEEC runtime manages actions specified at both the system and application level. Specifically, this case study uses the bodytrack and x264 applications from PARSEC. Both are modified to emit heartbeats and both are deployed with an initial goal of half the maximum performance achievable using all system resources.

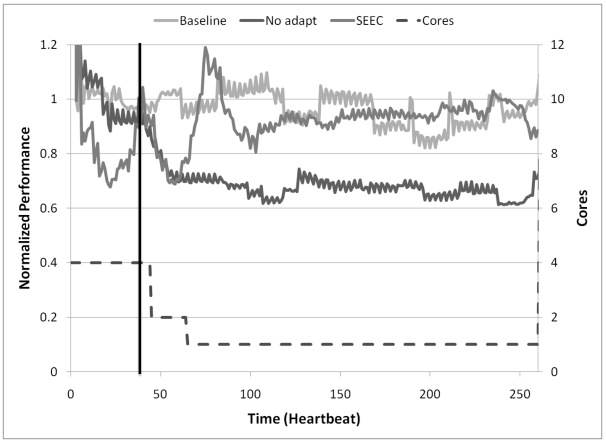
In this case there are two sets of actions available to the SEEC runtime. The first is changes to the number of cores assigned to each application as described in previous sections. Additionally, SEEC can alter x264's encoding algorithm. Using the systems programming interface, x264 is modified to specify 560 possible actions that alter the way it finds temporal redundancy between frames [18]. These actions increase speed at a cost of reduced video encoding quality. In this case, the SEEC runtime must create two separate models. The first captures bodytrack's response to cores, while the second captures x264's response to both core and algorithm changes. Additionally, x264 requests that SEEC favor system-level adaptations rather than application-level ones i.e. SEEC will only change x264's algorithm if it cannot meet x264's goals with the maximum compute resources.

We deploy both applications and the SEEC runtime system (using AL 1) with the processor set to 2.396 GHz. bodytrack is given a higher priority than x264. Then, 10% of the way through bodytrack's execution, we lower the processor speed to 1.596 GHz. The sudden change in frequency simulates a power cap or thermal throttling event and forces SEEC to adapt and manage a conflicting request for compute resources.

Figures 6(a) and 6(b) illustrate the behavior of SEEC in this scenario, where Figure 6(a) depicts bodytrack's response and Figure 6(b) shows that of x264. Both figures show performance (normalized to the average performance recorded for the application using four cores) on the left y-axis and the number of cores allocated to each application on the right y-axis. Time (measured in heartbeats) is shown on the x-axis. The time where frequency changes is shown by the solid vertical line in



(a) bodytrack



(b) x264

Figure 6: SEEC responding to clock speed changes.

each graph. For each application performance is shown for the baseline system with no clock frequency changes ("Baseline"), the system with clock frequency changes but no adaptation ("No adapt"), and SEEC adapting to clock frequency changes.

Figure 6(a) shows that SEEC maintains bodytrack's performance despite the loss in compute power. SEEC observes the performance loss as a reduction in heart rate and begins to deallocate cores from the lower priority x264 and assign them to bodytrack. As shown in the figure, without SEEC bodytrack would only achieve 65% of its desired performance, but with SEEC bodytrack is able to meet its goals.

Figure 6(b) shows that SEEC sacrifices x264's performance to meet the needs of bodytrack. SEEC deallocates cores from x264 but compensates for this loss by altering x264's algorithm. By managing both application and system level adaptations SEEC is able to resolve resource conflicts and meet both application's goals. We note that if x264 had been the higher priority application, SEEC would not have changed its algorithm because x264 requests system-level adaptations before application-level ones. In this case, SEEC would have assigned x264 more processors and bodytrack would not have met its goals.

This final case study demonstrates several aspects of SEEC. First, it shows how the SEEC runtime system can control multiple applications. Second, it shows how SEEC can simultaneously manage and coordinate actions specified at both the system and application level. Finally, it shows how SEEC can automatically adapt to fluctuations in the environment, in this case a sudden and unexpected change in processor speed. This behavior is possible because SEEC directly observes application performance and goals. We note that SEEC does not detect the clock frequency change directly, but instead detects a change in the applications' heart rates. Thus SEEC can respond to any change that alters the performance of the component applications.

5 Conclusion

This paper has presented the SEEC framework for self-aware computing. SEEC enables a new computational model where applications specify their goals, system software specifies possible actions, and the SEEC runtime dynamically selects actions to meet application goals. SEEC has three distinguishing features: 1) it decouples the concerns of application programmers from systems programmers while providing a unified framework for both, 2) it directly incorporates application goals, and 3) it uses a general, adaptive control-theoretic decision engine that is easily customized for specific needs. We have implemented several self-aware systems in the SEEC model and found it predictably achieves application-specified goals. Furthermore, the SEEC model is easily extended to a wide range of systems operating on different mechanisms. Finally, by observing and re-evaluating its decisions online, SEEC is able to adapt its own behavior. This flexibility to change decisions and take new actions allows SEEC to minimize power consumption while maintaining performance as well as respond to changes in the available compute resources. By predictably managing resource usage and responding to unforeseen events, we believe the SEEC model can reduce some of the application programmer's burden when working with modern multicore systems.

References

[1] Wattsup .net meter. http://www.wattsupmeters.com/.

[2] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36:49–58, December 2003.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *PLDI*, 2009.

[4] W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, June 2010.

[5] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg. Nap: a building block for remediating performance bottlenecks via black box network analysis. In *ICAC*, 2009.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, Oct 2008.

[7] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[8] A. Block, B. Brandenburg, J. H. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time system. In *ECRTS*, 2008.

[9] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. Jagr: An autonomous self-recovering application server. *AMS*, 0, 2003.

[10] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM J. Res. Dev.*, 50(2/3):239–248, 2006.

[11] F. Chang and V. Karamcheti. Automatic configuration and runtime adaptation of distributed applications. In *HPDC*, 2000.

[12] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *ISCA*, 2006.

[13] A. Goel, D. Steere, C. Pu, and J. Walpole. Swift: A feedback control and dynamic reconfiguration toolkit. In *2nd USENIX Windows NT Symposium*, 1998.

[14] C.-J. Hamann, M. Roitzsch, L. Reuther, J. Wolter, and H. Hartig. Probabilistic admission control to govern real-time systems under overload. In *ECRTS*, 2007.

[15] J. L. Hellerstein. Why feedback implementations fail: the importance of systematic testing. In *FeBID*, 2010.

[16] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[17] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*, 2010.

[18] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

[19] J. Hollingsworth and P. Keleher. Prediction and adaptation in active harmony. In *HPDC*, 1998.

[20] IBM Inc. IBM autonomic computing website. `http://www.research.ibm.com/autonomic/`, 2009.

[21] E. Ipek, O. Mutlu, J. F. Martnez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[22] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *HotOS*, Berkeley, CA, USA, 2005.

[23] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

[24] O. Krieger, M. Auslander, B. Rosenburg, R. W. J. W., Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *EuroSys*, 2006.

[25] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Processor power reduction via single-isa heterogeneous multicore architectures. *Computer Architecture Letters*, 2(1):2–2, Jan-Dec 2003.

[26] R. Laddaga. Guest editor's introduction: Creating robust software through self-adaptation. *IEEE Intelligent Systems*, 14:26–29, May 1999.

[27] W. Levine. *The control handbook*. CRC Press, 2005.

[28] B. Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632 –1650, Sept. 1999.

[29] C. Lu, Y. Lu, T. Abdelzaher, J. Stankovic, and S. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE TPDS*, 17(9):1014–1027, September 2006.

[30] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *RTSS*, 1999.

[31] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Controlling software applications via resource allocation within the heartbeats framework. In *CDC*, 2010.

[32] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, pages 19–25, Dec. 1995.

[33] S. Oberthür, C. Böke, and B. Griese. Dynamic online reconfiguration for customizable and self-optimizing operating systems. In *EMSOFT*, 2005.

[34] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[35] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: adaptive control of distributed applications. In *HPDC*, 1998.

[36] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys*, 2010.

[37] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[38] L. Sha, X. Liu, U. Y. Lu, and T. Abdelzaher. Queueing model based network server performance control. In *RTSS*, 2002.

[39] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys*, 2007.

[40] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, 2009.

[41] Q. Sun, G. Dai, and W. Pan. LPV model and its application in web server performance control. In *ICCSSE*, 2008.

[42] M. Tanelli, D. Ardagna, and M. Lovera. LPV model identification for power management of web service systems. In *MSC*, 2008.

[43] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *PPoPP*, 2005.

[44] G. Welch and G. Bishop. An introduction to the kalman filter. Technical Report TR 95-041, UNC Chapel Hill, Department of Computer Science.

[45] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. In *PACT*, 2010.

[46] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *ICDCS*. IEEE computer society, 2002.